REMARKS

In the Office Action dated May 6, 2004, claims 1-28 are pending and have been rejected. Accordingly, claims 1-28 are at issue. By virtue of the present Amendment, claims 1 and 20 have been amended. The support for this amendment can be found on page 5, lines 1-17 of the present application. With these amendments, no new matter has been added.

35 U.S.C. § 103

At section 7, claims 1-9, 13-22 and 26-28 are rejected under 35 U.S.C. § 103 as being anticipated by US Patent No. 5,790,664 ("*Coley*") in view of US Patent No 6,038,486 ("*Saitoh*").

The Examiner notes in the Office Action that "While the applicants arguments regarding what is shown in the art references and what is intended by the current application may be correct, the claims are written in a fashion so broad that they can be interpreted in many different ways". Applicants have revised independent claims 1 and 20 to further clarify the invention, stating that "searching *the communications network by a monitoring mechanism* for the installation site address" to more succinctly point out that the searching is done by a monitoring mechanism over the communications network. As discussed in the previous Amendment, neither *Coley* nor *Saitoh* discuss the searching for the installation site address over the network by a monitoring mechanism. Applicant respectfully requests that the Examiner enter these amendments as they are revised to comply with the Examiner's comments. Applicants believe that these changes in light of the Examiner's remarks and the discussion found in the previous Amendment place the claims in a form for allowance.

Claims 2-9, 13-19, 21, 22, and 26-28 are dependent from claims 1 and 20 and recite features not recited in claims 1 or 20. For reasons regarding claims 1 and 20 above, it is respectfully submitted that claims 2-9, 13-19, 21, 22 and 26-28 are also distinguishable over the cited *Coley* and *Saitoh* references.

35 U.S.C. § 112, First paragraph

At section 2 of the office action, claims 10, 11, 12, 23 and 24 are rejected under 35 USC § 112 first paragraph for containing subject matter that is not described in the specification in such a way to enable one skilled in the art to practice the invention. The Examiner states that "the specification says only 'a search device, such as a web spider, can be coded' but goes into no detail about how this code should work." The Examiner also "...asserts that this type of system of searching IP address or MAC addresses close to the address of known pieces of equipment is not well-known in the art..."

The Applicants respectfully disagree with the Examiner because the description in the present application is more than sufficient to enable on skilled in the art to practice the invention without undue experimentation. Claim 10 calls for "...searching a further site address based on the IP address in order to determine whether the product is also used at the further site address." Claims 11, 12, 23, and 24 contain similar language.

The searching for a site address is described in the specification on page 5 at lines 8-16 as follows:

> A search device **90**, similar to a web spider, which is used to search the world-wide web over the Internet, can be coded to take the information in the database **60** and search IP addresses close to the IP address, as listed in the list **64** in the database **60**, for additional units that have not been registered. The addresses close to the IP address are denoted by reference numeral **22**, and the additional units are denoted by reference numeral **12**. In that way, it is possible to determine whether the products are used in compliance with licenses. Moreover, the search device **90** can also use each

> of the MAC addresses that are assigned by the manufacturer to the products, as listed in the list **66**, to search for additional units based on the message **102**. If additional units are found, they can also be entered into the database **60**.

In the MPEP at 2164, the requirements for making a determination of enablement are discussed. There are several criteria here that demonstrate that the present invention is enabled.

> At 2164.01(b) of the MPEP, it states:
> As long as the specification discloses at least one method for making and using the claimed invention that bears a reasonable correlation to the entire scope of the claim, then the enablement requirement of 35 U.S.C. 112 is satisfied. *In re Fisher*, 427 F.2d 833, 839, 166 USPQ 18, 24 (CCPA 1970).

The claim calls for "...searching a further site address..." In the specification at page 5 lines 8-16, there is a description of several methods of how to search for site addresses. One such example (*In re Fisher* states that the requirement is only for a single method), "...the search device **90** can also use each of the MAC addresses that are assigned by the manufacturer to the products, as listed in the list **66**, to search for additional units based on the message **102.** " This description is a very explicit explanation of one method of "...searching a further site address..." as required for the claim. Searching through a predetermined list, such as "the MAC addresses that are assigned by the manufacturer to the products" is a simple task that is well known in the art. A freshman Computer Science student would know how write a simple loop to run through a fixed list trying to read memory from each device. This disclosure is clearly enabled.

In another example, the specification calls for a web-spider. An Internet-based search device, such as web spider, is well-known in the art. Attached to this Amendment is an article entitled "A Web Crawler in PERL" by Mike Thomas from Linux Journal (Volume 1997 Issue 40es) dated August 1997 that describes how to write a web spider. This article further includes source code for the implementation of a web spider. The

implementation of a web spider is well known and therefore this aspect of the claims is well enabled in the specification.

Furthermore, the standard that is used to determine enablement used in the Office Action is not correct. The Office Action states that "The issue is not whether one [of] ordinary skill would be able to program such a system, but that they would be able to program such a system without undue experimentation." This standard is not correctly applied here, for the MPEP at 2164.01 states:

> The fact that experimentation may be complex does not necessarily make it undue, if the art typically engages in such experimentation. *In re Certain Limited-Charge Cell Culture Microcarriers*, 221 USPQ 1165, 1174 (Int'l Trade Comm"n 1983), *aff'd. sub nom., Massachusetts Institute of Technology v. A.B. Fortia*, 774 F.2d 1104, 227 USPQ 428 (Fed. Cir. 1985). See also *In re Wands*, 858 F.2d at 737, 8 USPQ2d at 1404. The test of enablement is not whether any experimentation is necessary, but whether, if experimentation is necessary, it is undue. *In re Angstadt*, 537 F.2d 498, 504, 190 USPQ 214, 219 (CCPA 1976).

And further in 2164.06:

> The quantity of experimentation needed to be performed by one skilled in the art is only one factor involved in determining whether "undue experimentation" is required to make and use the invention. "[A]n extended period of experimentation may not be undue if the skilled artisan is given sufficient direction or guidance." *In re Colianni*, 561 F.2d 220, 224, 195 USPQ 150, 153 (CCPA 1977). " 'The test is not merely quantitative, since a considerable amount of experimentation is permissible, if it is merely routine, or if the specification in question provides a reasonable amount of guidance with respect to the direction in which the experimentation should proceed.' " *In re Wands*, 858 F.2d 731, 737, 8 USPQ2d 1400, 1404 (Fed. Cir. 1988) (citing *In re Angstadt*, 537 F.2d 489, 502-04, 190 USPQ 214, 217-19 (CCPA 1976)).

The question is whether the specification provides a reasonable amount of guidance with respect to the direction in which the experimentation should proceed. The present application provides at least three different areas of direction for one skilled in the art to use in implementing the present invention. The specification describes the use of a "web-crawler", an IP search, and a MAC address search. Routine coding of a search, either for

an IP address or a MAC address, are clearly not "undue experimentation" as explained by *In re Wands*.

In fact, the MPEP continues in 2164.08 "Nevertheless, not everything necessary to practice the invention need be disclosed. In fact, what is well-known is best omitted. *In re Buchner*, 929 F.2d 660, 661, 18 USPQ2d 1331, 1332 (Fed. Cir. 1991)". Enumerating specific search techniques that are well known in are best omitted, as has been done in the present application.

This is particularly important in the instant case because the present invention is not concerned with the programming of a search engine. Rather, the present invention uses an existing searching tool to search IP addresses over the Internet. General computer searching techniques have been well known since the 1950s, and web-spiders have been well known in the industry for at least 7 years.

Therefore, applicant requests that the Examiner remove the rejections of claims 10-12, 23 and 24 based upon 35 USC 112 first paragraph.

Conclusion

In view of the foregoing Amendment and Remarks, Applicants respectfully submit that claims 1-28 claim matter that is distinct from the prior art and request that the objections and rejections be withdrawn. With the submission of this Amendment, this application is in condition for further examination and early consideration of the claims at issue and early allowance is hereby requested. The Commissioner is authorized to charge any additional fees or credit any overpayments associated with this Amendment to Deposit Account 19-2875 (SAA-49). Applicants further invite the Examiner to contact

the undersigned representative at the telephone number below to discuss any matters pertaining to the present Application.

Respectfully submitted,

By: _____

Richard A. Baker
Patent Agent, Reg. No. 48,124

SCHNEIDER ELECTRIC AUTOMATION BUSINESS
1415 South Roselle Road
Palatine, IL 60067
Telephone: 978/975-9789
Facsimile: 847/925-7419

A Web Crawler in Perl

Here's how spiders search the Web collecting information for you.

by Mike Thomas

Web-crawling robots, or spiders, have a certain mystique among Internet users. We all use search engines like Lycos and Infoseek to find resources on the Internet, and these engines use spiders to gather the information they present to us. Very few of us, however, actually use a spider program directly.

Spiders are network applications which traverse the Web, accumulating statistics about the content found. So how does a web spider work? The algorithm is straightforward:

1. Create a queue of URLs to be searched beginning with one or more known URLs.
2. Pull a URL out of the queue and fetch the Hypertext Markup Language (HTML) page which can be found at that location.
3. Scan the HTML page looking for new-found hyperlinks. Add the URLs for any hyperlinks found to the URL queue.
4. If there are URLs left in the queue, go to step 2.

Listing 1 is a program, spider.pl, which implements the above algorithm in Perl. This program should run on any Linux system with Perl version 4 or higher installed. Note that all code mentioned in this article assumes Perl is installed in /usr/bin/Perl. These scripts are available for download on my web page at http://www.javanet.com/~thomas/.

To run the spider at the shell prompt use the command:

```
spider.pl <starting-URL<search-phrase>
```

The spider will commence the search. The starting URL must be fully specified, or it may not parse correctly. The spider searches the initial page and all its descendant pages for the given search phrase. The URL of any page with a match is printed. To print a list of URLs from the SSC site containing the phrase "*Linux Journal*", type:

```
spider.pl http://www.ssc.com/ "Linux Journal"
```

The Perl variable `$DEBUG`, defined in the first few lines of spider.pl, is used to control the amount of output the spider produces. `$DEBUG` can range from 0 (matching URLs are printed) to 2 (status of the program and dumps of internal data structures are output).

Interaction with the Internet

The most interesting thing about the spider program is the fact that it is a network program. The subroutine `get_http()` encapsulates all the network programming required to implement a spider; it does the "fetch" alluded to in step 2 of the above algorithm. This subroutine opens a socket to a server and uses the HTTP protocol to retrieve a page. If the server has a port number appended to it, this port is used to establish the connection; otherwise, the well-known port 80 is used.

Once a connection to the remote machine has been established, `get_http()` sends a string such as:

```
GET /index.html HTTP/1.0
```

This string is followed by two newline characters. This is a snippet of the Hypertext Transport Protocol (HTTP), the protocol on which the Web is based. This request asks the web server to which we are connected to send the contents of the file /index.html to us. `get_http()` then reads the socket until an end of file is encountered. Since HTTP is a connectionless protocol, this is the extent of the conversation. We submit a request, the web server sends a response and the connection is terminated.

The response from the web server consists of a header, as specified by the HTTP standard, and the HTML-tagged text making up the page. These two parts of the response are separated by a blank line. Running the spider at debug level 2 will display the HTTP headers for you as a page is fetched. The following is a typical response from a web server.

```
HTTP/1.0 200 OK
Date: Tue, 11 Feb 1997 21:54:05 GMT
Server: Apache/1.0.5
Content-type: text/html
Content-length: 79
Last-modified: Fri, 22 Nov 1996 10:11:48 GMT

<HTML><TITLE>My Web Page</TITLE>
<BODY>
This is my web page.
</BODY>
</HTML>
```

The spider program checks the `Content-type` field in the HTTP header as it arrives. If the content is of any MIME type other than text/html or text/plain, the download is aborted. This avoids the time-consuming download of things like .Z and .tar.gz files, which we don't wish to search. While most sites use the FTP protocol to transfer this type of file, more and more sites are using HTTP.

There is a hardware dependency in `get_http()` that you should be aware of if you are running Linux on a SPARC or Alpha. When building the network addresses for the socket, the Perl `pack()` routine is used to encode integer data. The line:

```
$sockaddr="S n a4 x8";
```

is suitable only for 32-bit CPUs. To get around this, see Mike Mull's article ``Perl and Sockets'' in *LJ* Issue 35.

The URL Queue

Once the spider has downloaded the HTML source for a web page, we can scan it for text matching the search phrase and notify the user if we find a match.

We can also find any hypertext links embedded in the page and use them as a starting point for a further search. This is exactly what the spider program does; it scans the HTML content for anchor tags of the form `<A HREF="url">` and adds any links it finds to its queue of URLs.

A hyperlink in an HTML page can be in one of several forms. Some of these must be combined with the URL of the page in which they're embedded to get a complete URL. This is done by the `fqURL()` function. It combines the URL of the current page and the URL of a hyperlink found in that page to

produce a complete URL for the hyperlink.

For example, here are some links which might be found in a fictitious web page at http://www.ddd.com/clients/index.html, together with the resulting URL produced by `fqURL()`.

URL in Anchor Tag	Resulting URL
http://www.eee.org/index.html	http://www.eee.org/index.html
att.html	http://www.ddd.com/clients/att.html
/att.html	http://www.ddd.com/att.html

As these examples show, the spider can handle both a fully-specified URL and a URL with only a document name. When only a document name is given, it can be either a fully qualified path or a relative path. In addition, the spider can handle URLs with port numbers embedded, e.g., http://www.ddd.com:1234/index.html.

One function not implemented in `fqURL()` is the stripping of back-references (../) from a URL. Ideally, the URL /test/.../index.html is translated to /index.html, and we know that both point to the same document.

Once we have a fully-specified URL for a hyperlink, we can add it to our queue of URLs to be scanned. One concern that crops up is how to limit our search to a given subset of the Internet. An unrestricted search would end up downloading a good portion of the world-wide Internet content--not something we want to do to our compadres with whom we share network bandwidth. The approach spider.pl takes is to discard any URL that does not have the same host name as the beginning URL; thus, the spider is limited to a single host. We could also extend the program to specify a set of legal hosts, allowing a small group of servers to be searched for content.

Another issue that arises when handling the links we've found is how to prevent the spider from going in circles. Circular hyperlinks are very common on the Web. For example, page A has a link to page B, and page B has a link back to page A. If we point our spider at page A, it finds the link to B and checks it out. On B it finds a link to A and checks it out. This loop continues indefinitely. The easiest way to avoid getting trapped in a loop is to keep track of where the spider has been and ensure that it doesn't return. Step 2 in the algorithm shown at the beginning of this article suggests that we "pull a URL out of our queue" and visit it. The spider program doesn't remove the URL from the queue. Instead, it marks that URL as having been scanned. If the spider later finds a hyperlink to this URL, it can ignore it, knowing it has already visited the page. Our URL queue holds both visited and unvisited URLs.

The set of pages the spider has visited will grow steadily, and the set of pages it has yet to visit can grow and and shrink quickly, depending on the number of hyperlinks found in each page. If a large site is to be traversed you may need to store the URL queue in a database, rather than in memory as we've done here. The associative array that holds the URL queue, `%URLqueue`, could easily be linked to a GDBM database with the Perl 4 functions `dbmopen()` and `dbmclose()` or Perl 5 functions `tie()` and `untie()`.

Responsible Use

Note that you should not unleash this beast on the Internet at large, not only because of the bandwidth it consumes, but also because of Internet conventions. The document request the spider sends is a one line `GET` request. To strictly follow the HTTP protocol, it should also include `User-Agent` and `From` fields, giving the remote server the opportunity to deny our request and/or collect statistics.

This program also ignores the ``robots.txt'' convention that is used by administrators to deny access to robots. The file /robots.txt should be checked before any further scanning of a host. This file indicates if scanning from a robot is welcome and declares any subdirectories that are off-limits. A robots.txt file that excludes scanning of only 2 directories looks like this:

```
Useuagent: *
Disallow: /tmp/
Disallow: /cgi-bin/
```

A file that prohibits all scanning on a particular web server looks like this:

```
User-agent: *
Disallow: /
```

Robots like our spider can place a heavy load on a web server, and we don't wish to use it on servers that have been declared off-limits to robots by their administrators

Application of the spider.pl Script

How might we use the spider program, other than as a curiosity? One use for the program would be as a replacement for one of the web site index and query programs like Harvest (http://harvest.cs.colorado.edu/Harvest/) or Excite for Web Servers (http://www.excite.com/navigate/prodinfo.html). These programs are large and complicated. They often provide the functionality of the Perl spider program, a means of archiving the text retrieved and a CGI query engine to run against the resulting database. Ongoing maintenance is required, since the query engine runs against the database rather than against the actual site content; therefore, the database must be regenerated whenever a change is made to the content of the site.

Some search engines, such as Excite for Web Servers, cannot index the content at a remote site. These engines build their database from the files which make up the web site, rather than from data retrieved across a network. If you had two web sites whose content was to appear in a single search application, these tools would not be appropriate. Furthermore, the Linux version of Excite for Web Servers is still in the ``coming soon'' stage.

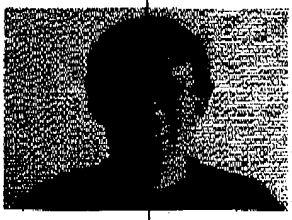
Listing 2 and Listing 3 show a simple CGI search engine that is implemented using the spider.pl program. Listing 2 is an HTML form which calls spiderfind.cgi to process its input. Listing 3 is spiderfind.cgi. It first uses Brigitte Jellinek's library to move the data entered in the form into an associative array. It then calls the spider.pl program using the Perl `system()` function and passes the form data as parameters. Finally, it converts the output from spider.pl into a series of HTML links. The user's browser will display a list of hyperlinked URLs in which the search text was found. Note that the name of the host to search is specified by a hidden field in the HTML document. There are better and more security-conscious ways for two Perl programs to interact than through a Perl `system()` call, but I wanted to use an unmodified copy of spider.pl for this demonstration.

This script doesn't provide the complete functionality of the packages mentioned above, and it won't perform as well. Since we're doing the search against web server documents across the Net, we don't have the advantage of index files; therefore, the search will be slower and more processor-intensive. However, this script is easy to install and easier to maintain than those engines.

Another application that could be built using the spider.pl program is a broken link scanner for the Web. The HTTP response we showed previously began with the line ```HTTP/1.0 200 OK`'', indicating the

request could be fulfilled. If we tried to hit a URL with a non-existent document, we would get the line "HTTP/1.0 404 Not found" instead. We could use this as an indication that the document does not exist and print the URL which referenced this page.

The modifications to the spider program needed to accomplish this are minor. Every time a hyperlink's URL is added to the URL queue, we also record the URL of the document in which we found the hyperlink. Then, when the spider checks out the hyperlink and receives a "404 Not found" response, it outputs the URL of the referring page.



Mike Thomas is an Internet application developer working for a consulting firm in Saskatchewan, Canada. Mike lives in Massachusetts and uses two Linux systems to telecommute 2000 miles to his job and to Graduate School at the University of Regina. He can be reached by e-mail at thomas@javanet.com.

[| Magazine Table of Contents]

[Magazine Table of Contents]

```
#! /usr/bin/perl
#
# spider.pl    Set tabstops to 3.
#

$| = 1;
# 0=no debug, 1-display progress, 2-complete dump
$DEBUG = 0;
# Check hyperlinks to other hosts?
$SPANHOST$ = "off";

if(scalar(@ARGV) < 2){
print "Usage: $0 <fully-qualified-URL> <search-phrase>\n";
exit 1;
}

# Initialize.
%URLqueue = ();
chop($client_host=`hostname`);
$been = 0;
$search_phrase = $ARGV[1];

# Load the queue with the first URL to hit.
$URLqueue{$ARGV[0]} = 0;
$thisURL = &find_new(%URLqueue);

# While there's a URL in our queue which we haven't looked at ...
while($thisURL ne ""){

# Progress report.
$count = 0;
while(($key,$value) = each(%URLqueue)){
        $count ++;
}
print "-----------------------------------------\n" if($DEBUG>=1);
printf("Been: %d  To Go: %d\n", $been, $count-$been)
if($DEBUG>=1);
print "Current URL: $thisURL\n" if($DEBUG>=1);
&dump_stack() if($DEBUG>=2);

# Split the protocol from the URL.
($protocol, $rest) = $thisURL =~ m|^([^:/]*):(.*)$|;

# If the protocol is http, fetch the page and process it.
if($protocol eq "http"){

        # Split out the hostname, port and document.
        ($server_host, $port, $document) =
                $rest =~ m|^//([^:/]*):*([0-9]*)/*([^:]*)$|;

        # Get the page of text and remove CR/LF characters and HTML
        # comments from it.
        $page_text = &get_http($client_host, $server_host, $port,
                $document);
        $page_text =~ tr/\r\n//d;
        $page_text =~ s|<!--[^>]*-->||g;

        # Report if our search string is found here.
        if($page_text =~ m|$search_phrase|i){
                print "$thisURL\n"
```

```
        }

        # Find anchors in the HTML and update our list of URLs..
        (@anchors) = $page_text =~ m|<A[^>]*HREF\s*=\s*"([^
">]*)"|gi;
        foreach $anchor (@anchors){
                $newURL = &fqURL($thisURL, $anchor);
                if($URLqueue{$newURL} > 0){

                        # Increment the count for URLs we've already
                        # checked out.
                        $URLqueue{$newURL}++;

                }else{

                        # Add a zero record for URLs we haven't
                        # encountered.
                        # Optionally, ignore URL's which point to other
                        # hosts.
                        ($new_host) =
                           $newURL =~ m|^[^:/]*://([^/:]*):*[0-9]*/*[^:]*$|;
                        if($SPANHOSTS eq "on" || $new_host eq
                                $server_host){
                                $URLqueue{$newURL}=0;
                        }
                }
        }
}else{
        print "Protocol '$protocol' ignored.\n" if($DEBUG>=1);
}

# Record the fact that we've been here, and get a new URL to process.
$URLqueue{$thisURL} ++;
$been ++;
$thisURL = &find_new(%URLqueue);

}
exit;

#-------------------------------------------------------------
# Build a fully specified URL.
#-------------------------------------------------------------
sub fqURL
{
local($thisURL, $anchor) = @_;
local($has_proto, $has_lead_slash, $currprot, $currhost, $newURL);

# Strip anything following a number sign '#', because its
# just a reference to a position within a page.
$anchor =~ s|^.*#[^#]*$|$1|;

# Examine anchor to see what parts of the URL are specified.
$has_proto = 0;
$has_lead_slash=0;
$has_proto = 1 if($anchor =~ m|^[^/:]+:|);
$has_lead_slash = 1 if ($anchor =~ m|^/|);

if($has_proto == 1){

        # If protocol specified, assume anchor is fully qualified.
```

```
        $newURL = $anchor;

    }
    elsif($has_lead_slash == 1){

        # If document has a leading slash, it just needs protocol and host.
        ($currprot, $currhost) = $thisURL =~ m|^([^:/]*):/+([^:/]*)|;
        $newURL = $currprot . "://" . $currhost . $anchor;

    }
    else{

        # Anchor must be just relative pathname, so append it to current URL.
        ($newURL) = $thisURL =~ m|^(.*)/[^/]*$|;
        $newURL .= "/" if (! ($newURL =~ m|/$|));
        $newURL .= $anchor;

    }
    if($DEBUG >=2){
        print "Link Found\n   In:$thisURL\n   Anchor:$anchor\n   Result: $newURL\n"
    }
    return $newURL;
    }

    #-------------------------------------------------------------
    # Do a linear search of the URL stack to find a URL with a data
    # value of 0 (i.e. one we haven't checked out yet).
    #-------------------------------------------------------------
    sub find_new
    {
    local(%URLqueue) = @_;
    local($key, $value);

    while(($key, $value) = each(%URLqueue)){
        return $key if($value == 0);
    }
    return "";
    }

    #-------------------------------------------------------------
    # Debugging utility.
    #-------------------------------------------------------------
    sub dump_stack
    {
    local($key, $x);
    local($done, $togo) = ("", "");

    foreach $key (keys(%URLqueue)){
        if($URLqueue{$key} == 0){
            $togo .= "  " . $key . "\n";
        }else{
            $done .= "  " . $key . " (hitcount = "
                . $URLqueue{$key} . ")\n";
        }
    }

    print "Been There:\n" . $done;
    print "To Go:\n" . $togo;
    print "------- Hit Q to Quit, Enter to Continue -------\n";
    read(STDIN, $key, 1);
```

```
exit(1) if($key eq 'Q' || $key eq 'q');
}

#--------------------------------------------------------------------
# Get the page indicated by the $server_host and $document parameters.
#--------------------------------------------------------------------
sub get_http
{
local($client_host, $server_host, $port, $document) = @_;
local($name,$aliases,$type,$len);
local($this,$thisaddr,$that,$thataddr);
local($client_host, $sockaddr, $a,$b,$c,$d);
local($page, $header, $header_text, $content);

# Some constants used to access the TCP network.
$AF_INET=2;
$SOCK_STREAM=1;

# Use default http port if none specified.
$port = 80 if($port == 0);

# Get the protocol number for TCP.
($name,$aliases,$proto)=getprotobyname("tcp");

# Get the IP addresses for the two hosts.
($name,$aliases,$type,$len,$thisaddr) = gethostbyname($client_host);
($name,$aliases,$type,$len,$thataddr) = gethostbyname($server_host);

# Check we could resolve the server host name.
($a,$b,$c,$d) = unpack('C4', $thataddr);
if($a eq "" && $b eq "" && $c eq  "" && $d eq ""){
        print "ERROR: Unknown host $server_host.\n";
        return "";
}
print "Server: $server_host ($a.$b.$c.$d)\n" if($DEBUG>=2);

# Pack the AF_INET magic number, the port, and the (already packed) IP
# addresses into the same format as the C structure would use. Note
# this is architecture dependent: this pack format works for 32 bit
# architectures.
$sockaddr="S n a4 x8";
$this=pack($sockaddr, $AF_INET, 0, $thisaddr);
$that=pack($sockaddr, $AF_INET, $port, $thataddr);

# Create the socket and connect.
if(socket(S, $AF_INET, $SOCK_STREAM, $proto) == false){
        print "ERROR: Cannot create socket.\n";
        return "";
}
print "Socket OK\n" if($DEBUG>=2);
if(connect(S, $that) == false){
        print "ERROR: Cannot connect to server $server_host,
                port $port.\n";
        return "";
}
print "Connect OK\n" if($DEBUG>>>>>>>>=2);

# Turn buffering in the socket off, and send request to the server.
select(S); $| = 1; select(STDOUT);
print S "GET /$document HTTP/1.0\n\n";
```

```
# Receive the response. Check to ensure the response is of MIME
# type text/html or text/plain.
$page = "";
$header = 1;
$header_text = "";
while(<S>){

        # Check if we've hit the end of the HTTP header (an empty
line).
        # If we have, check for a content-type header line, and
ensure
        # it is valid.
        if( m|^[\n\r]*$| ){
                $header = 0;
                ($content) = $header_text =~ m|Content-type: (\S+)|i;
                if($content ne "text/html" && $content ne "text/plain"){
                        print "Content type '$content' ignored.\n"
                                if($DEBUG>=1);
                        last;
                }
        }
        # Save to a header string if we're still working on the HTTP
        # header.
        elsif($header == 1){
                $header_text .= "    " . $_;
        }
        # Otherwise, save to the html page string.
        else{
                $page .= $_;
        }

print "HTTP header: \n $header_text" if($DEBUG>=2);
return $page;
}
```

```
#!/usr/bin/perl
#
# spiderfind.cgi
#
# Note: must set $DEBUG=0 in spider.pl.

$| = 1;

# Use Brigitte Jellinek's library to get form
# data into the array %form_data.
require("./bjellis.pl");
&GetFormArgs();

$search = $form_data{"search"};
$url = $form_data{"url"};

# Build a command using the data passed from the
# form. Note the quotes around the data from the
# form are vital. They prevent a web user from
# entering a search string like
# "test; cd /; rm-r *"
# and deleting every file the web server user has
# access to.
$cmd = sprintf('./spider.pl "%s" "%s"', $form_data{"url"},
        $form_data{"search"});

# Run the command and wrap the results up in HTML
# and print it back to the web server.
$result = `$cmd`;
print "Content-type: text/html\n\n";
print "<HTML><TITLE>Search Results</TITLE>\n";
print "<BODY><H2>Search Results for '$search' "
print "on '$url'</H2>\n";
print "</BODY></HTML>";
$result =~ s|([^\n]*)\n|
        <A href="$1">$1</A><BR>\n|g;
print $result;
```

```
<HTML>
<TITLE> Mike's WebCrawler Demo </TITLE>

<BODY>
<H2> Mike's WebCrawler Demo</H2>
<P> This form will invoke Mike's webcrawler to
search for the phrase you enter in the form below.
This is a live search, so if the connection to the
selected host is slow, the response may take some
time.</P>

<FORM ACTION="spiderfind.cgi" METHOD=POST>
        Search String: <INPUT TYPE="TEXT"i
                NAME="search" SIZE=40></BR>
        <INPUT TYPE="HIDDEN" NAME="url"
                VALUE="http://www.ssc.com/">
        <INPUT TYPE=SUBMIT>
</FORM>

</BODY>
</HTML>
```